

# *MultiMarkdown v6 Quick Start Guide*

*Fletcher T. Penney*

*April 29, 2017*

## *Contents*

<i>Introduction</i>	<i>1</i>
<i>Performance</i>	<i>1</i>
<i>Parse Tree</i>	<i>3</i>
<i>Features</i>	<i>4</i>
<i>Abbreviations (Or Acronyms)</i>	<i>4</i>
<i>Citations</i>	<i>4</i>
<i>CriticMarkup</i>	<i>5</i>
<i>Emph and Strong</i>	<i>5</i>
<i>EPUB 3 Support</i>	<i>5</i>
<i>Fenced Code Blocks</i>	<i>6</i>
<i>Footnotes</i>	<i>6</i>
<i>Glossary Terms</i>	<i>6</i>
<i>Internationalization</i>	<i>6</i>
<i>LaTeX Changes</i>	<i>7</i>
<i>Metadata</i>	<i>8</i>
<i>Table of Contents</i>	<i>8</i>
<i>Transclusion</i>	<i>8</i>
<i>Future Steps</i>	<i>9</i>

## *Introduction*

Version: 6.0.1

This document serves as a description of MultiMarkdown (MMD) v6, as well as a sample document to demonstrate the various features. Specifically, differences from MMD v5 will be pointed out.

## *Performance*

A big motivating factor leading to the development of MMD v6 was performance. When MMD first migrated from Perl to C (based

on peg-markdown<sup>1</sup>), it was among the fastest Markdown parsers available. That was many years ago, and the “competition” has made a great deal of progress since that time.

<sup>1</sup> <https://github.com/jgm/peg-markdown>

When developing MMD v6, one of my goals was to keep MMD at least in the ballpark of the fastest processors. Of course, being *the* fastest would be fantastic, but I was more concerned with ensuring that the code was easily understood, and easily updated with new features in the future.

MMD v3 – v5 used a PEG to handle the parsing. This made it easy to understand the relationship between the MMD grammar and the parsing code, since they were one and the same. However, the parsing code generated by the parsers was not particularly fast, and was prone to troublesome edge cases with terrible performance characteristics.

The first step in MMD v6 parsing is to break the source text into a series of tokens, which may consist of plain text, whitespace, or special characters such as ‘\*’, ‘[’, etc. This chain of tokens is then used to perform the actual parsing.

MMD v6 divides the parsing into two separate phases, which actually fits more with Markdown’s design philosophically.

1. Block parsing consists of identifying the “type” of each line of the source text, and grouping the lines into blocks (e.g. paragraphs, lists, blockquotes, etc.) Some blocks are a single line (e.g. ATX headers), and others can be many lines long. The block parsing in MMD v6 is handled by a parser generated by lemon<sup>2</sup>. This parser allows the block structure to be more readily understood by non-programmers, but the generated parser is still fast.
2. Span parsing consists of identifying Markdown/MMD structures that occur inside of blocks, such as links, images, strong, emph, etc. Most of these structures require matching pairs of tokens to specify where the span starts and where it ends. Most of these spans allow arbitrary levels of nesting as well. This made parsing them correctly in the PEG-based code difficult and slow. MMD v6 uses a different approach that is accurate and has good performance characteristics even with edge cases. Basically, it keeps a stack of each “opening” token as it steps through the token chain. When a “closing” token is found, it is paired with the most recent appropriate opener on the stack. Any tokens in between the opener and closer are removed, as they are not able to be matched any more. To avoid unnecessary searches for non-existent openers, the parser keeps track of which opening tokens have been discovered. This allows the parser to continue moving forwards without having to go backwards and re-parse any previously vis-

<sup>2</sup> <http://www.hwaci.com/sw/lemon/>

ited tokens.

The result of this redesigned MMD parser is that it can parse short documents more quickly than CommonMark<sup>3</sup>, and takes only 15% – 20% longer to parse long documents. I have not delved too deeply into this, but I presume that CommonMark has a bit more “set-up” time that becomes expensive when parsing a short document (e.g. a paragraph or two). But this cost becomes negligible when parsing longer documents (e.g. file sizes of 1 MB). So depending on your use case, CommonMark may well be faster than MMD, but we’re talking about splitting hairs here. . . . Recent comparisons show MMD v6 taking approximately 4.37 seconds to parse a 108 MB file (approximately 24.8 MB/second), and CommonMark took 3.72 seconds for the same file (29.2 MB/second). For comparison, MMD v5.4 took approximately 94 second for the same file (1.15 MB/second).

<sup>3</sup> <http://commonmark.org/>

For a more realistic file of approx 28 kb (the source of the Markdown Syntax web page), both MMD and CommonMark parse it too quickly to accurately measure. In fact, it requires a file consisting of the original file copied 32 times over (0.85 MB) before `/usr/bin/env time` reports a time over the minimum threshold of 0.01 seconds for either program.

There is still potentially room for additional optimization in MMD. However, even if I can’t close the performance gap with CommonMark on longer files, the additional features of MMD compared with Markdown in addition to the increased legibility of the source code of MMD (in my biased opinion anyway) make this project worthwhile.

### *Parse Tree*

MMD v6 performs its parsing in the following steps:

1. Start with a null-terminated string of source text (C style string)
2. Lex string into token chain
3. Parse token chain into blocks
4. Parse tokens within each block into span level structures (e.g. strong, emph, etc.)
5. Export the token tree into the desired output format (e.g. HTML, LaTeX, etc.) and return the resulting C style string

**OR**

6. Use the resulting token tree for your own purposes.

The token tree (AST) includes starting offsets and length of each token, allowing you to use MMD as part of a syntax highlighter. MMD v5 did not have this functionality in the public version, in part because the PEG parsers used did not provide reliable offset positions, requiring a great deal of effort when I adapted MMD for use in MultiMarkdown Composer<sup>4</sup>.

<sup>4</sup> <http://multimarkdown.com/>

These steps are managed using the `mmd_engine` “object”. An individual `mmd_engine` cannot be used by multiple threads simultaneously, so if `libMultiMarkdown` is to be used in a multithreaded program, a separate `mmd_engine` should be created for each thread. Alternatively, just use the slightly more abstracted `mmd_convert_string()` function that handles creating and destroying the `mmd_engine` automatically.

## Features

### *Abbreviations (Or Acronyms)*

This file includes the use of MMD as an abbreviation for MultiMarkdown. The abbreviation will be expanded on the first use, and the shortened form will be used on subsequent occurrences.

Abbreviations can be specified using inline or reference syntax. The inline variant requires that the abbreviation be wrapped in parentheses and immediately follows the `>`.

```
[>MMD] is an abbreviation. So is [>(MD) Markdown].
```

```
[>MMD]: MultiMarkdown
```

There is also a “shortcut” method for abbreviations that is similar to the approach used in prior versions of MMD. You specify the definition for the abbreviation in the usual manner, but MMD will automatically identify each instance where the abbreviation is used and substitute it automatically. In this case, the abbreviation is limited to a more basic character set which includes letters, numbers, periods, and hyphens, but not much else. For more complex abbreviations, you must explicitly mark uses of the abbreviation.

### *Citations*

Citations can be specified using an inline syntax, just like inline footnotes.

The HTML output for citations now uses parentheses instead of brackets, e.g. (1) instead of [1].

## CriticMarkup

MMD v6 has improved support for CriticMarkup<sup>5</sup>, both in terms of parsing, and in terms of support for each output format. You can insert text, ~~delete text~~, substitute one thing for another, **highlight text**, and in the text.

<sup>5</sup> <http://criticmarkup.com/>

leave comments

If you don't specify any command line options, then MMD will apply special formatting to the CriticMarkup formatting as in the preceding paragraph. Alternatively, you can use the `-a\`-accept or `-r\`-reject options to cause MMD to accept or reject, respectively, the proposed changes within the CM markup. When doing this, CM will work across blank lines. Without either of these two options, then CriticMarkup that spans a blank line is not recognized as such. I'm working on options for this for the future.

## Emph and Strong

The basics of emphasis and strong emphasis are unchanged, but the parsing engine has been improved to be more accurate, particularly in various edge cases where proper parsing can be difficult.

## EPUB 3 Support

MMD v6 now provides support for direct creation of EPUB 3<sup>6</sup> files. Previously a separate tool was required to create EPUB files from MMD. It's now built-in. Currently, EPUB 3 files are built using the usual HTML 5 output. No extra CSS is applied, so the default from the reader will be used. Images are not yet supported, but are planned for the future.

<sup>6</sup> <https://en.wikipedia.org/wiki/EPUB>

EPUB files can be highly customized with other tools, and I recommend doing that for production quality files. For example, apparently performance is improved when the content is divided into multiple files (e.g. one file per chapter). MMD creates EPUB 3 files using a single file. Tools like Sigil<sup>7</sup> are useful for improving your EPUB files, and I recommend doing that.

<sup>7</sup> <https://sigil-ebook.com/>

Not all EPUB readers support v3 files. I don't plan on adding support for older versions of the EPUB format, but other tools can convert to other document formats you need. Same goes for Amazon's ebook formats – the Calibre<sup>8</sup> program can also be used to interconvert between formats.

<sup>8</sup> <https://calibre-ebook.com/>

**NOTE:** Because EPUB documents are binary files, MMD only creates them when run in batch mode (using the `-b\`-batch options). Otherwise, it simply outputs the HTML 5 file that would serve as the primary content for the EPUB.

### *Fenced Code Blocks*

Fenced code blocks are fundamentally the same as MMD v5, except:

1. The leading and trailing fences can be 3, 4, or 5 backticks in length. That should be sufficient to account for complex documents without requiring a more complex parser.
2. If there is no trailing fence, then everything after the leading fence is considered to be part of the code block.

### *Footnotes*

The HTML output for footnotes now uses superscripts instead of brackets, e.g. `<sup>1</sup>` instead of `[1]`.

### *Glossary Terms*

If there are terms in your document you wish to define in a glossary at the end of your document, you can define them using the glossary syntax.

Glossary terms can be specified using inline or reference syntax. The inline variant requires that the abbreviation be wrapped in parentheses and immediately follows the `?`.

`[?(glossary) The glossary collects information about important terms used in your document]` is a glossary term.

`[?glossary]` is also a glossary term.

`[?glossary]:` The glossary collects information about important terms used in your document

Much like abbreviations, there is also a “shortcut” method that is similar to the approach used in prior versions of MMD. You specify the definition for the glossary term in the usual manner, but MMD will automatically identify each instance where the term is used and substitute it automatically. In this case, the term is limited to a more basic character set which includes letters, numbers, periods, and hyphens, but not much else. For more complex glossary terms, you must explicitly mark uses of the term.

### *Internationalization*

MMD v6 includes support for substituting certain text phrases in other languages. This only affects the HTML format.

## *LaTeX Changes*

LaTeX support is slightly different than in prior versions of MMD. It is designed to be a bit more consistent, and easier for basic use.

The previous approach used two types of metadata:

- `latex input` – this uses the name of a latex file that will be used in a `\input{file}` command. This key can be used multiple times (the only metadata key that worked this way), and all the basic metadata is written to the LaTeX file in order.
- `latex footer` – this file worked the same way as `latex input`, but was inserted at the end of the file

In practice, one typically needs to be able to insert `\input` commands at only a few key places in the final document:

1. At the very beginning
2. After metadata, and before the body of the document
3. After the body of the document

MMD 6 standardizes the metadata to use 3 new keys:

1. `latex leader` – this specifies a file that will be used at the very beginning of the document.
2. `latex begin` – this comes after metadata, and before the body of the document. This will usually include the `\begin{document}` command, hence the name.
3. `latex footer` – this comes after the body of the document.

You can use these 3 keys to replace the old `latex input` metadata keys, as long as you pay attention as to which is which. If you used more than three include statements, you may have to combine your latex files to fit into the new system.

***In addition***, there is a new shortcut key – `latex config`. This allows you to specify a “document name” that is used to automatically identify the corresponding `latex leader`, `latex begin`, and `latex footer` files. For example, using `latex config: article` is the same as using:

```
latex leader: mmd6-article-leader
latex begin: mmd6-article-begin
latex footer: mmd6-article-footer
```

Using the new system will require migrating your old configuration to the new naming convention, but once done I believe it should be much more intuitive to use.

The LaTeX support files included with the MMD v6 repository support the use of the following `latex config` values by default:

- `article`
- `tufte-book`
- `tufte-handout`

**\*\*NOTE:** You do have to install the MMD support files into the proper location **\*\***for your system. I would like to make this easier, but haven't found the **\*\*best configuration yet.**

### *Metadata*

Metadata in MMD v6 includes new support for LaTeX – the `latex config` key allows you to automatically setup of multiple `latex include` files at once. The default setups that I use would typically consist of one LaTeX file to be included at the top of the file, one to be included right at the beginning of the document, and one to be included at the end of the document. If you want to specify the `latex` files separately, you can use `latex leader`, `latex begin`, and `latex footer`.

There are new metadata keys for controlling internationalization:

- `language` – specify the content language for a document, using the two letter code for the language (e.g. `en` for English). Where possible, this will also set the default `quotes language`.
- `quotes language` – specify which variant of smart quotes to use. Valid options are `dutch`, `french`, `german`, `germanguillemets`, `swedish`, `nl`, `fr`, `de`, `sv`. Anything else defaults to English.

### *Table of Contents*

By placing `{{T0C}}` in your document, you can insert an automatically generated Table of Contents in your document. As of MMD v6, the native Table of Contents functionality is used when exporting to LaTeX or OpenDocument formats.

### *Transclusion*

File transclusion works basically the same way – `{{file}}` is used to indicate a file that needs to be transcluded. `{{file.*}}` allows



for wildcard transclusion. What's different is that the way search paths are handled is more flexible, though it may take a moment to understand.

When you process a file with MMD, it uses that file's directory as the search path for included files. For example:

Directory	Transcluded Filename	Resolved Path
/foo/bar/	bat	/foo/bar/bat
/foo/bar/	baz/bat	/foo/bar/baz/bat
/foo/bar/	../bat	/foo/bat

This is the same as MMD v 5. What's different is that when you transclude a file, the search path stays the same as the "parent" file, **UNLESS** you use the `transclude base` metadata to override it. The simplest override is:

```
transclude base: .
```

This means that any transclusions within the file will be calculated relative to the file, regardless of the original search path.

Alternatively you could specify that any transclusion happens inside a subfolder:

```
transclude base: folder/
```

Or you can specify an absolute path:

```
transclude base: /some/path
```

This flexibility means that you can transclude different files based on whether a file is being processed by itself or as part of a "parent" file. This can be useful when a particular file can either be a standalone document, or a chapter inside a larger document.

## *Future Steps*

Some features I plan to implement at some point:

1. OPML export support is not available in v6. I plan on adding improved support for this at some point. I was hoping to be able to re-use the existing v6 parser but it might be simpler to use the approach from v5 and earlier, which was to have a separate parser tuned to only identify headers and "stuff between headers".

